Constraint Satisfaction Problems
Enforcing Consistency

Bernhard Nebel, Julien Hué, and Stefan Wölfl

Albert-Ludwigs-Universität Freiburg

May 9/21/23, 2012
1 Arc Consistency

2 Path Consistency

3 Higher Levels of $i$-Consistency

4 Extensions of Arc Consistency
Enforcing consistency

- The more explicit and tight constraint networks are, the more restricted is the search space of partial solutions.
- **Idea**: infer new constraints without “removing” (by methods called local consistency enforcing, bounded consistency inference, constraint propagation).
- Consistency-enforcing algorithms aim at *assisting search*: How can we extend a given partial solution of a small subnetwork to a partial solution of a larger subnetwork?
1 Arc Consistency
Some useful conventions

- In what follows we will always assume that the variables of a constraint network appear in some order.

- Further, we assume that $C$ does not contain unary constraints, i.e., constraints in $C$ are always relations with arity $n > 1$, but we allow that the domains $D_i$ are possibly empty. This is no restriction, since we can rewrite $D_i$:

$$D_i \leftarrow D_i \cap R_{v_i}$$

and then remove $R_{v_i}$ from the network.

$D_i$ will be referred to as domains, unary constraint, or domain constraint.

- We write constraints with scheme $(v_i, \ldots, v_j, \ldots v_k)$ in the form $R_{i\ldots j\ldots k}$. 
Arc Consistency

**Arc consistency**

Let \( N = \langle V, D, C \rangle \) be a constraint network.

**Definition**

(a) A variable \( v_i \) is arc-consistent relative to variable \( v_j \) if for each value \( a_i \in D_i \), there exists an \( a_j \in D_j \) with \( (a_i, a_j) \in R_{ij} \) (in case that \( R_{ij} \) exists in \( C \)).

(b) An “arc constraint” \( R_{ij} \) is arc-consistent if \( v_i \) is arc-consistent relative to \( v_j \) and \( v_j \) is arc-consistent rel. to \( v_i \).

(c) A network \( N \) is arc-consistent if all its arc constraints are arc-consistent.

**Lemma**

Checking whether a network \( N = \langle V, D, C \rangle \) is arc-consistent requires at most \( e \cdot k^2 \) operations (where \( e \) is the number of its binary constraints and \( k \) is an upper bound of its domain sizes).
Example

Consider a constraint network with two variables $v_1$ and $v_2$, domains $D_1 = D_2 = \{1, 2, 3\}$, and the binary constraint expressed by $v_1 < v_2$.

Figure: A network that is not arc-consistent
Revising a single domain

\textbf{Revise} (v_i, v_j):

\textit{Input:} a network with two variables v_i, v_j, domains D_i and D_j, and constraint R_{ij}

\textit{Result:} a network with refined D_i such that v_i is arc-consistent relative to v_j

\begin{verbatim}
for each \(a_i \in D_i\)
    \textbf{if} there is no \(a_j \in D_j\) with \((a_i, a_j) \in R_{ij}\)
    \textbf{then} remove \(a_i\) from \(D_i\)
endif
endfor
\end{verbatim}

This is equivalent to applying:

\[D_i \leftarrow D_i \cap \pi_i(R_{ij} \bowtie D_j)\]
Revising a single domain

**Lemma**

*The complexity of Revise is $O(k^2)$, where $k$ is an upper bound of the domain sizes.*

Note: With a simple modification of the Revise algorithm one could improve to $O(t)$, where $t$ is the maximal number of tuples occurring in one of the binary constraints in the network.
Enforcing arc consistency: AC1

**AC1**(\(N\)):

- **Input:** a constraint network \(N = \langle V, D, C \rangle\)
- **Result:** \(N\) arc-consistent, but equivalent to input network

```
repeat
  for each arc \(\{v_i, v_j\}\) with \(R_{ij} \in C\)
    Revise\((v_i, v_j)\)
    Revise\((v_j, v_i)\)
  endfor
until no domain is changed
```
Enforcing arc consistency: AC1

Lemma
Let $N$ be a constraint network with $n$ variables, each with a domain of size $\leq k$, and $e$ binary constraints.
Applying AC1 on the network runs in time $O(e \cdot n \cdot k^3)$.

Proof.
One cycle through all binary constraints takes $O(e \cdot k^2)$. In the worst case, one cycle just removes one value from one domain. Moreover, there are at most $n \cdot k$ values. This results in an upper bound of $O(e \cdot n \cdot k^3)$. □

Note: If the input network is already arc-consistent, then AC1 runs in time $O(e \cdot k^2)$. 
Example: AC1

Consider a constraint network with three variables \( v_1, v_2, \) and \( v_3 \), domains \( D_1 = D_2 = \{1, 2, 3\} \), and the binary constraints expressed by \( v_1 < v_2 \) and \( v_2 < v_3 \).

Note: Enforcing arc consistency may already be sufficient to show that a constraint network is inconsistent. For example, add the constraint \( v_3 < v_1 \) to the network just considered.
Enforcing arc consistency: AC3

Idea: no need to process all constraints if only a few domains have changed. Operate on a queue of constraints to be processed.

**AC3(N):**

*Input:* a constraint network \( N = \langle V, D, C \rangle \)

*Result:* an equivalent, but arc-consistent network

**for** each pair \( v_i, v_j \) that occurs in a constraint \( R_{ij} \)

\[ \text{queue} \leftarrow \text{queue} \cup \{(v_i, v_j), (v_j, v_i)\} \]

**endfor**

**while** queue is not empty

**select** and **remove** \((v_i, v_j)\) from queue

**Revise**(\( v_i, v_j \))

**if** Revise\((v_i, v_j)\) changes \(D_i\)

\[ \text{then} \quad \text{queue} \leftarrow \text{queue} \cup \{(v_k, v_i) : k \neq i, k \neq j\} \]

**endif**

**endwhile**
Enforcing arc consistency: AC3

Lemma

Let $N$ be a constraint network with $n$ variables, each with a domain of size $\leq k$, and $e$ binary constraints.

Applying AC3 on the network runs in time $O(e \cdot k^3)$.

Proof.

Consider a single constraint. Each time, when it is reintroduced into the queue, the domain of one of its variables must have been changed. Since there are at most $2 \cdot k$ values, AC3 processes each constraint at most $2 \cdot k$ times. Because we have $e$ constraints and processing of each is in time $O(k^2)$, we obtain $O(e \cdot k^3)$.

Note: If the input network is arc-consistent, then AC3 runs in time $O(e \cdot k^2)$.\qed
Enforcing arc consistency: AC3

Example: Consider a constraint network with 3 variables $v_1$, $v_2$, $v_3$ with domains $D_1 = \{2, 4\}$ and $D_2 = D_3 = \{2, 5\}$, and two constraints expressed by $v_3 \mid v_1$ and $v_3 \mid v_2$ ("divides").
Enforcing arc consistency: AC4

- To verify that a network is arc-consistent needs \( e \cdot k^2 \) operations.
- The following algorithm AC4 achieves optimal performance, ...
- at the cost of “best case performance”, which is \( \Omega(e \cdot k^2) \).

Idea:
- Associate to each value \( a_i \) in the domain of variable \( v_i \) the amount of support from variable \( v_j \) (i.e., the number of values in \( D_j \) that are consistent with \( a_i \));
- remove a value \( a_i \) if it loses support from any other variable

Details:
- \( Q \): queue of unsupported variable-value pairs;
- \( \text{counter}(v_i, a_i, v_j) \): amount of support for \( a_i \) from \( v_j \);
- \( S[v_j, a_j] \): set containing variable-value pairs \((v_i, a_i)\) (with \( i \neq j \)) supported by \((v_j, a_j)\).
Enforcing arc consistency: AC4

\textbf{AC4}(N):

\textit{Input:} a constraint network \( N = \langle V, D, C \rangle \)

\textit{Result:} an equivalent, but arc-consistent network

\( Q \leftarrow \emptyset; \)
\( S[v_j, a_j] \leftarrow \emptyset, \) \( \text{counter}(v_i, a_i, v_j) \leftarrow 0 \) for all \( R_{ij} \in C, a_i \in D_i, a_j \in D_j \)

\textbf{for} each \( R_{ij} \in C, a_i \in D_i \)

\textbf{for} each \( a_j \in D_j \)

\textbf{if} \( (a_i, a_j) \in R_{ij} \) \textbf{then}

increment \( \text{counter}(v_i, a_i, v_j) \) and add \( (v_i, a_i) \) to \( S[v_j, a_j] \)

\textbf{if} \( \text{counter}(v_i, a_i, v_j) = 0 \) \textbf{then}

add \( (v_i, a_i) \) to \( Q \) and remove \( a_i \) from \( D_i \)

\textbf{while} \( Q \) is not empty

select and remove \( (v_j, a_j) \) from \( Q \)

\textbf{for} each \( (v_i, a_i) \) in \( S[v_j, a_j] \)

\textbf{if} \( a_i \in D_i \) \textbf{then}

decrement \( \text{counter}(v_i, a_i, v_j) \)

\textbf{if} \( \text{counter}(v_i, a_i, v_j) = 0 \) \textbf{then}

add \( (v_i, a_i) \) to \( Q \) and remove \( a_i \) from \( D_i \)
Example: AC4

Consider the same network as for AC3.
Constraints: $v_3 \vdash v_1$ and $v_3 \vdash v_2$.

The initialization steps yield:

\[
\begin{align*}
S[v_3, 2] &= \{ (v_1, 2), (v_1, 4), (v_2, 2) \} \\
S[v_3, 5] &= \{ (v_2, 5) \} \\
S[v_2, 2] &= \{ (v_3, 2) \} \\
S[v_2, 5] &= \{ (v_3, 5) \} \\
S[v_1, 2] &= \{ (v_3, 2) \} \\
S[v_1, 4] &= \{ (v_3, 2) \}
\end{align*}
\]
**Example: AC4**

The initialization steps yield:

\[
S[v_3, 2] = \{(v_1, 2), (v_1, 4), (v_2, 2)\} \quad S[v_3, 5] = \{(v_2, 5)\}
\]
\[
S[v_2, 2] = \{(v_3, 2)\} \quad S[v_2, 5] = \{(v_3, 5)\}
\]
\[
S[v_1, 2] = \{(v_3, 2)\} \quad S[v_1, 4] = \{(v_3, 2)\}
\]

Furthermore:

\[
\text{counter}(v_3, 2, v_1) = 2 \quad \text{and} \quad \text{counter}(v_3, 5, v_1) = 0.
\]

All other counters are 1 (note: we only need consider counters between connected variables).

\[
Q = \{(v_3, 5)\} \quad \text{and} \quad D_3 = \{2\}.
\]

When \((v_3, 5)\) is selected (and removed) from \(Q\), we obtain \(\text{counter}(v_2, 5, v_3) = 0\). \((v_2, 5)\) is added to \(Q\) and 5 deleted from \(D_2\). Then \((v_2, 5)\) is selected from \(Q\). \((v_2, 5)\) has only support for \((v_3, 5)\), but 5 has already been removed from \(D_3\), ...
Enforcing arc consistency: AC2001

- **Fine-grained** algorithms (like AC4) directly propagate the removal of a value \((v_i, a_i)\) to values \((v_j, a_j)\) which were supported by \((v_i, a_i)\)
- ... while **coarse-grained** algorithms (like AC3) propagate changes on the level of the domains only
- Nevertheless coarse-grained algorithms have advantages: no need for additional data structures \(S[v_j, a_j]\) (costs for initialization and maintenance)
- **AC2001** is a coarse-grained method: works like AC3, but with a different revise function: achieves optimal run time \(O(e \cdot k^2)\).
Revise2001 in AC2001

- Assume orderings on each of the domains (use dummy value \textit{nil} smaller than all domain values)
- AC2001 first initializes and maintains pointers $Last(v_i, a_i, v_j) \leftarrow \textit{nil}$

\textbf{Revise2001}(v_i, v_j):

\textit{Input:} a network with two variables $v_i, v_j$, domains $D_i$ and $D_j$, and constraint $R_{ij}$

\textit{Result:} a network with a refined domain $D_i$

\begin{itemize}
  \item \textbf{for} each $a_i$ in $D_i$ with $Last(v_i, a_i, v_j) \notin D_j$
  \item $a_j \leftarrow$ the smallest value $a$ in $D_j$ with $a > Last(v_i, a_i, v_j)$ and $(a_i, a) \in R_{ij}$
  \item \textbf{if} $a_j$ \textbf{exists} \textbf{then}
  \item $Last(v_i, a_i, v_j) \leftarrow a_j$
  \item \textbf{else}
  \item remove $a_i$ from $D_i$
\end{itemize}
Path Consistency

2 Path Consistency
Beyond arc consistency

- Sometimes “enforcing arc consistency” is sufficient for detecting inconsistent (unsolvable) networks; but . . .
- enforcing arc consistency is not complete for deciding consistency of networks; because . . .
- inferences rely only on domain constraints and single binary constraints defined on the domains.

⇒ We consider further concepts of local consistency
Path consistency

Definition

(a) A binary constraint $R_{ij}$ for variables $v_i, v_j$ is path-consistent relative to a third variable $v_k$ if for every pair $(a_i, a_j) \in R_{ij}$, there exists an $a_k \in D_k$ such that $(a_i, a_k) \in R_{ik}$ and $(a_k, a_j) \in R_{kj}$.

(b) A pair of distinct variables $v_i, v_j$ is path-consistent relative to variable $v_k$ if any instantiation $a$ of $\{v_i, v_j\}$ with $(a(v_i), a(v_j)) \in R_{ij}$ can be extended to an instantiation $a'$ of $\{v_i, v_j, v_k\}$ such that $(a'(v_i), a'(v_k)) \in R_{ik}$ and $(a'(v_k), a'(v_j)) \in R_{kj}$ (“extended” means: $a = a'|_{\{v_i,v_j\}}$).

(c) A set of distinct variables $\{v_i, v_j, v_k\}$ is path-consistent if any pair of these variables is path-consistent relative to the omitted third variable.

(d) A constraint network is path-consistent if all its three-element subsets of variables are path-consistent.
An example

**Figure:** This network is arc-consistent, but not path-consistent.
Revising a path

**Revise-3**$(\{v_i, v_j\}, v_k)$:

**Input:** a binary network $\langle V, D, C \rangle$ with variables $v_i, v_j, v_k$

**Result:** a revised constraint $R_{ij}$ path-consistent with $v_k$

foreach each pair $(a_i, a_j) \in R_{ij}$

if there is no $a_k \in D_k$ such that $(a_i, a_k) \in R_{ik}$

and $(a_j, a_k) \in R_{jk}$

then remove $(a_i, a_j)$ from $R_{ij}$

endif

dofor

This is equivalent to applying:

$$R_{ij} \leftarrow R_{ij} \cap \pi_{ij}(R_{ik} \Join D_k \Join R_{kj})$$
Revising a path: Properties

Lemma

When applied to a constraint network \( N \), procedure \( \text{Revise-3}(\{v_i, v_j\}, v_k) \):

- does not do anything if the pair \( v_i, v_j \) is path-consistent relative to \( v_k \), and otherwise
- transforms the network into an equivalent form where the pair \( v_i, v_j \) is path-consistent relative to \( v_k \).

Proof.

From the definition of path consistency.
Revising a path: Complexity

Lemma

Let \( t \) be the maximal number of tuples in one of the binary constraints, and let \( k \) be an upper bound for the domain sizes.

The worst-case runtime of Revise-3 is \( \mathcal{O}(t \cdot k) \).

The best-case runtime of Revise-3 is \( \Omega(t) \).

With respect to \( k \), the complexity of Revise-3 can also be expressed as \( \mathcal{O}(k^3) \) in the worst and \( \Omega(k^2) \) in the best case.
**Enforcing path consistency: PC1**

\[ \text{PC1}(N): \]

*Input:* a constraint network \( N = \langle V, D, C \rangle \)

*Result:* an equivalent, path-consistent network

repeat
  for each (ordered) triple of variables \( v_i, v_j, v_k \):
    Revise-3(\( \{v_i, v_j\}, v_k \))
  endfor
until no constraint is changed
Enforcing path consistency: Soundness of PC1

Lemma

*When applied to a constraint network $N$, the PC1 algorithm computes a path-consistent constraint network which is equivalent to $N$.**

Proof.

Follows directly from the properties of Revise-3.
Enforcing path consistency: Complexity of PC1

Lemma
Let $N$ be a constraint network with $n$ variables, each with a domain of size $\leq k$. Let $t$ be an upper bound of the number of tuples in one of the binary constraints in $C$.

The worst-case runtime of PC1 on this network is $O(n^5 \cdot t^2 \cdot k)$. The best-case runtime of PC1 on this network is $\Omega(n^3 \cdot t)$.

The runtime bounds can also be stated as $O(n^5 \cdot k^5)$ and $\Omega(n^3 \cdot k^2)$, respectively.
Enforcing path consistency: Complexity of PC1

Proof (worst case).
In each iteration of the outer loop in PC1, only one value pair might be removed from one of the constraints. Hence the number of iterations may be as large as $O(n^2 \cdot t)$.
Processing a specific triple of constraints (there are $O(n^3)$ many such triples) costs $O(t \cdot k)$.
Hence each iteration costs $O(n^3 \cdot t \cdot k)$.

Proof (best case).
In the best case, the network is already path-consistent and only one iteration through the outer loop is needed. There are $\Omega(n^3)$ calls to Revise-3, each requiring time $\Omega(t)$ in the best case.
Enforcing path consistency: PC2

**PC2(N):**

*Input:* a constraint network $N = \langle V, D, C \rangle$

*Result:* an equivalent, path-consistent network $N'$

```
queue ← \{(i, k, j) : 1 ≤ i < j ≤ n, 1 ≤ k ≤ n, k ≠ i, k ≠ j\}

while queue is not empty
    select and remove a triple $(i, k, j)$ from queue
    Revise-3($\{v_i, v_j\}, v_k$)
    if $R_{ij}$ has changed then
        queue ← queue ∪ \{(l, i, j), (l, j, i) : 1 ≤ l ≤ n, l ≠ i, j\}
    endif
endwhile
```
Enforcing path consistency: Soundness of PC2

Lemma

When applied to a constraint network $N$, the PC2 algorithm computes a path-consistent constraint network which is equivalent to $N$.

Proof.

Equivalence follows directly from the properties of Revise-3. To see that the remaining constraint network is path-consistent, verify the following invariant:

Before and after each iteration of the while-loop, for each pair $v_i, v_j$ which is not path-consistent relative to $v_k$, one of the triples $(i, k, j)$ and $(j, k, i)$ is contained in the queue.
Lemma
Let $N$ be a constraint network with $n$ variables, each with a domain of size $\leq k$. Let $t$ be an upper bound of the number of tuples in one of the binary constraints in $N$.

The worst-case runtime of PC2 on this network is $O(n^3 \cdot t^2 \cdot k)$.
The best-case runtime of PC2 on this network is $\Omega(n^3 \cdot t)$.

Because of $t \leq k^2$, the runtime bounds can also be stated as $O(n^3 \cdot k^5)$ and $\Omega(n^3 \cdot k^2)$, respectively.
Enforcing path consistency: Complexity of PC2

Proof (worst case).
There are initially $O(n^3)$ elements in the queue. Whenever some constraint $R_{ij}$ is reduced, which can happen at most $O(n^2 \cdot t)$ many times, $O(n)$ elements are added to the queue. Thus, the total number of elements added to the queue is bounded by $O(n^3 \cdot t)$.
Each iteration of the while loop removes an element from the queue, so there are at most $O(n^3 \cdot t)$ iterations and hence at most $O(n^3 \cdot t)$ calls to Revise-3, each requiring time $O(t \cdot k)$, for a total runtime bound of $O(n^3 \cdot t^2 \cdot k)$.

Proof (best case).
Similar to PC1.
## Arc and path consistency: Overview

<table>
<thead>
<tr>
<th></th>
<th>Worst Case</th>
<th>Best Case</th>
</tr>
</thead>
<tbody>
<tr>
<td>AC1</td>
<td>$O(n \cdot k \cdot e \cdot t)$</td>
<td>$\Omega(e \cdot k)$</td>
</tr>
<tr>
<td>AC3</td>
<td>$O(e \cdot k \cdot t)$</td>
<td>$\Omega(e \cdot k)$</td>
</tr>
<tr>
<td>AC4</td>
<td>$O(e \cdot k^2)$</td>
<td>$\Omega(e \cdot k^2)$</td>
</tr>
<tr>
<td>PC1</td>
<td>$O(n^5 \cdot t^2 \cdot k)$</td>
<td>$\Omega(n^3 \cdot t)$</td>
</tr>
<tr>
<td>PC2</td>
<td>$O(n^3 \cdot t^2 \cdot k)$</td>
<td>$\Omega(n^3 \cdot t)$</td>
</tr>
<tr>
<td>PC4*</td>
<td>$O(n^3 \cdot t \cdot k)$</td>
<td>$\Omega(n^3 \cdot t \cdot k)$</td>
</tr>
</tbody>
</table>

*not discussed in this lecture

**Remark:** $O(n^3 \cdot t \cdot k)$ is the optimal (worst-case) runtime for enforcing path consistency, i.e., there are (arbitrarily large) constraint networks for which no better algorithm exists.
3 Higher Levels of $i$-Consistency
Higher levels of $i$-consistency

The local consistency notions presented so far can be roughly summarized as follows:

- **Arc consistency**: Every consistent assignment to a single variable can be consistently extended to any second variable.

- **Path consistency**: Every consistent assignment to two variables can be consistently extended to any third variable.

(Side remark: This is a bit of an oversimplification because we ignored $k$-ary constraints with $k \geq 3$ so far.)

It is easy to see that the general idea of local consistency can be readily extended to larger variable sets.
Let $N = \langle V, D, C \rangle$ be a constraint network.

**Definition**

(a) A relation $R_S \in C$ with scope $S$ of size $i - 1$ is *i-consistent* relative to variable $v_i \notin S$ if for every tuple $t \in R_S$, there exists an $a \in D_i$ such that $(t, a)$ is consistent.

(b) A constraint network is *i-consistent* if any consistent instantiation of $i - 1$ (distinct) variables $v_1, \ldots, v_{i-1}$ of the network can be extended to a *consistent* instantiation of the variables $v_1, \ldots, v_i$, where $v_i$ is any variable in $V$ distinct from $v_1, \ldots, v_{i-1}$.
Global consistency

Definition

- A network $N$ is strongly $i$-consistent if it is $j$-consistent for each $j \leq i$.
- A network $N$ with $n$ variables is globally consistent if it is strongly $n$-consistent.

Note: Solutions to globally consistent networks can be found without search. (How?)
Arc/path consistency vs. 2/3-consistency

Note:

- 2-consistency coincides with arc consistency.
- For networks containing binary constraints only, 3-consistency coincides with path consistency.
- Each 3-consistent network is path-consistent.
- The converse is not true: For networks with constraints of arity $\geq 3$, 3-consistency is ** stricter** than path consistency.
**3-Consistency: Examples**

Example

\[ V = \{ v_1, v_2, v_3 \} \]
\[ D_1 = D_2 = D_3 = \{ 0, 1 \} \]
\[ R_{123} = \{(0, 0, 0)\} \]

Example

\[ V = \{ v_1, v_2, v_3 \} \]
\[ D_1 = D_2 = D_3 = \{ 0, 1 \} \]
\[ R_{123} = \{(0, 0, 0), (0, 1, 1), (1, 0, 1), (1, 1, 0)\} \]
\[ R_{12} = R_{13} = R_{23} = \{(0, 1), (1, 0), (1, 1)\} \]
Revise-\(i\)

\[\text{Revise-}i(\{v_1, \ldots, v_{i-1}\}, v_i):\]

\textbf{Input:} a network \( \langle V, D, C \rangle \) and a constraint \( R_S \) with scope \( S = \{v_1, \ldots, v_{i-1}\} \)

\textbf{Result:} a constraint \( R_S \) which is \( i \)-consistent rel. to \( v_i \)

for each instantiation \( \bar{a}_{i-1} \in R_S \)

if there is no \( a_i \in D_i \) such that \( (\bar{a}_{i-1}, a_i) \) is consistent

then remove \( \bar{a}_{i-1} \) from \( R_S \)

endif

endfor

- \( R_S \) can be the universal relation wrt. \( S \).
- If the input network is binary, then Revise-\(i\) runs in time \( O(k^i) \).
- In general, Revise-\(i\) runs in time \( O((2 \cdot k)^i) \), since \( O(2^i) \) constraints must be processed for each tuple.
**i-Consistency: Algorithm**

**Enforce i-Consistency** ($N$):

*Input:* a constraint network $N = \langle V, D, C \rangle$.
*Result:* an $i$-consistent network equivalent to $N$.

repeat
  for each subset of $S \subseteq V$ of size $i - 1$ and each $v_i \notin S$
    Revise-$i$($\{v_1, \ldots, v_{i-1}\}, v_i$)
  endfor
until no constraint is changed

The Revise-$i$ call can equivalently be stated as follows:
Let $S$ be the set of all subsets of $\{v_1, \ldots, v_i\}$ that contain $v_i$ and occur as scopes of some constraint in the network. Then apply

$$R_S \leftarrow R_S \cap \pi_S(\bigwedge_{S' \in S} R_{S'}).$$
Lemma

Let $N$ be a constraint network with $n$ variables, each with a domain of size $\leq k$. When applied to $N$, the “Enforce $i$-Consistency” algorithm runs in time $O(2^i \cdot (n \cdot k)^{2i-1})$.

Proof.

Each call to Revise-$i$ requires time $O((2 \cdot k)^i)$. In each iteration of the outer loop, $O(n^i)$ combinations of $S$ and $v_i$ need to be processed. If only one tuple is removed from one constraint in each iteration up to the final one, the outer loop may need to iterate $O(n^{i-1} \cdot k^{i-1})$ times. This leads to an overall runtime of $O(2^i \cdot (n \cdot k)^{2i-1})$.

Note: Improvements similar to AC4 and PC4 exist and achieve a worst-case runtime of $O(n^i \cdot k^i)$. 


### $i$-Consistency: Comparison to ACx and PCx

<table>
<thead>
<tr>
<th>$i$-consistency, $i = 2$</th>
<th>Worst Case</th>
</tr>
</thead>
<tbody>
<tr>
<td>$i$-consistency, $i = 2$</td>
<td>$O(n^3 \cdot k^3)$</td>
</tr>
<tr>
<td>AC1</td>
<td>$O(n \cdot k \cdot e \cdot t) = O(n^3 \cdot k^3)$</td>
</tr>
<tr>
<td>AC3</td>
<td>$O(e \cdot k \cdot t) = O(n^2 \cdot k^3)$</td>
</tr>
<tr>
<td>AC4</td>
<td>$O(n^2 \cdot k^2)$</td>
</tr>
<tr>
<td>improved $i$-consistency*, $i = 2$</td>
<td></td>
</tr>
<tr>
<td>$O(n^2 \cdot k^2)$</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>$i$-consistency, $i = 3$</th>
<th>Worst Case</th>
</tr>
</thead>
<tbody>
<tr>
<td>$i$-consistency, $i = 3$</td>
<td>$O(n^5 \cdot k^5)$</td>
</tr>
<tr>
<td>PC1</td>
<td>$O(n^5 \cdot t^2 \cdot k) = O(n^5 \cdot k^5)$</td>
</tr>
<tr>
<td>PC2</td>
<td>$O(n^3 \cdot t^2 \cdot k) = O(n^3 \cdot k^5)$</td>
</tr>
<tr>
<td>PC4*</td>
<td>$O(n^3 \cdot k^3)$</td>
</tr>
<tr>
<td>improved $i$-consistency*, $i = 3$</td>
<td>$O(n^3 \cdot k^3)$</td>
</tr>
</tbody>
</table>

*not discussed in this lecture

**Remark:** $O(n^i \cdot k^i)$ is the optimal (worst-case) runtime for enforcing $i$-consistency, i.e., there are (arbitrarily large) constraint networks for which no better algorithm exists.
4 Extensions of Arc Consistency
Extensions of Arc consistency

- General \(i\)-consistency is powerful, but expensive to enforce.
- Usually, arc consistency and path consistency offer a good compromise between pruning power and computational overhead.
- However, they are of limited usefulness for constraints on more than two variables.

**Example**

Consider a constraint network with three integer variables \(v_1, v_2, v_3 \geq 0\) and the constraints \(v_3 \geq 13\) and \(v_1 + v_2 + v_3 \leq 15\). We should be able to infer \(v_1 \leq 2\) and \(v_2 \leq 2\), but regular arc consistency is not enough!

\(\Rightarrow\) Consider generalizations of arc consistency to non-binary constraints.
Generalized arc consistency

Let $N = \langle V, D, C \rangle$ be a constraint network.

Definition

(a) A variable $v_i$ is (generalized) arc-consistent relative to a constraint $(S, R) \in C$ with $v_i$ in $S = (v_1, \ldots, v_n)$ if for every value $a_i \in D_i$ there exists a tuple $\bar{a} \in R \cap (D_1 \times \cdots \times D_n)$ with $\bar{a}[i] = a_i$, i.e.,

$$D_i \subseteq \pi_i(R \cap (D_1 \times \cdots \times D_n)).$$

(b) A constraint $(S, R) \in C$ is (generalized) arc-consistent if all variables in its scope $S$ are generalized arc-consistent relative to $R$.

(c) A network $N$ is (generalized) arc-consistent if all its constraints are generalized arc-consistent.
Generalized arc consistency: Update rule

To enforce generalized arc consistency, repeatedly apply

\[ D_i \leftarrow D_i \cap \pi_i(R_S \bowtie D_S \setminus \{v_i\}) \]

Note how this generalizes the usual arc consistency update rule:

\[ D_i \leftarrow D_i \cap \pi_i(R_{ij} \bowtie D_j) \]
Alternatives to generalized arc consistency

- Like arc consistency, generalized arc consistency propagates constraints by considering a single constraint at a time.
- In particular, it considers how assignments to each individual variable are restricted by the values allowed for the other variables participating in the constraint.
- Alternatively, we can consider how each individual variable restricts the values allowed for the other variables participating in the constraint:

\[
R_S\backslash\{v_i\} \leftarrow R_S\backslash\{v_i\} \cap \pi_S\backslash\{v_i\}(R_S \bowtie D_i)
\]

(relational arc consistency)
- Note that in the case of binary constraints, these two cases are the same, so both approaches are natural generalizations of (binary) arc consistency.
Generalizations of arc consistency: Comparison

\[
\text{AC: } D_i \leftarrow D_i \cap \pi_i(R_{ij} \Join D_j)
\]

\[
\text{generalized AC: } D_i \leftarrow D_i \cap \pi_i(R_S \Join D_{S\setminus\{v_i\}})
\]

\[
\text{relational AC: } R_{S\setminus\{v_i\}} \leftarrow R_{S\setminus\{v_i\}} \cap \pi_{S\setminus\{v_i\}}(R_S \Join D_i)
\]

Example

Consider a constraint network with three integer variables \(v_1, v_2, v_3 \geq 0\) and the constraints \(v_3 \geq 13\) and \(v_1 + v_2 + v_3 \leq 15\).

- Generalized AC infers \(v_1 \leq 2, v_2 \leq 2\).
- Relational AC infers \(v_1 + v_2 \leq 2\).
Literature

Christian Bessiere.
Constraint propagation,
Chapter 3 of Handbook of Constraint Programming, 2006

Rina Dechter.
Constraint Processing,
Chapter 3, Morgan Kaufmann, 2003

Alan K. Mackworth.
Constraint satisfaction.

Alan K. Mackworth.
Consistency in networks of relations.

Ugo Montanari.
Networks of constraints: fundamental properties and applications to picture processing.